МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ МІСЬКОГО ГОСПОДАРСТВА ІМЕНІ О. М. БЕКЕТОВА

І. Л. Яковицький

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

«ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ»

(для студентів освітньо-кваліфікаційного рівня бакалавр у галузі знань 12 - Інформаційні технології за спеціальністю 122 – Комп'ютерні науки та інформаційні технології)

Харків – ХНУМГ ім. О.М. Бекетова – 2017

Яковицький І. Л. Конспект лекцій з дисципліни «Об’єктно-орієнтоване програмування» (для студентів освітньо-кваліфікаційного рівня бакалавр у галузі знань 12 – Інформаційні технології за спеціальністю 122 – Комп’ютерні науки та інформаційні технології / І.Л. Яковицький ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків: ХНУМГ ім. О. М. Бекетова, 2017. – 39 с.

Автор – канд. техн. наук, доц. І. Л. Яковицький

Рецензент д-р фіз.-мат. наук, проф. кафедри мікроелектроніки, електронних приладів та пристроїв Харківського національного університету радіоелектроніки О. В. Грицунов.

Рекомендовано кафедрою прикладної математики і інформаційних технологій, протокол № 2 від 29.08.2015 р.

Зміст

Перелік прикладів.

Парадигма ООП і можливості С ++

Об’ектно-орієнтоване програмування та мова С++

Об’єктно-орієнтоване програмування (ООП) сучасна методологія програмування. Вона є результатом п’ятидесятирічного досвіду і практики, які беруть початок в мові Simula67 і тривають в Smalltalk, LISP, Clu і більш пізніх - Actor, Eiffel, Objective C, C++, Java.

ООП формує стиль програмування, який описує поведінку реального світу так, що деталі розробки приховані. Це дозволяє тому, хто переймається тим, міркувати у термінах, властивих цьому завданні, а не програмування.

Мову С++ було створено на початку 80-х років XX ст. Створив її Бйорн Страуструп. Він ставив наступні цілі: (1) не втратити сумісності мови С++ із мовою С та (2) розширити мову програмування С конструкціями, які спираються на концепції обєктно-орієнтованого програмування, серед яких центральне місце належить поняттю клас, що було започатковано у мові Simula 67.

Мову С розробив Денніс Річі на початку 70-х рр. XX ст., як мову системного програмування для операційної системи UNIX. Поступово вона отримала популярність як мова програмування загального призначення.

Кінцева мета створення С++ надати професійному програмістові мову, яку можна використовувати при створенні об’єктно-орієнтованого програмного забезпечення, без втрати ефективності або переносимості мови С. Перші кроки зробив Денніс Річі, його наступником стали Бйорн Страуструп і зростаюче співтовариство програмістів-практиків.

Об’єктно-орієнтоване програмування

ООП сфокусовано на описі даних і поведінки, які нерозривно пов’язані. Такий опис називають КЛАС, а об’єкти є екземплярами класу. Наприклад, многочлен має область значень, яку змінюють операції додавання і множення многочленів.

ООП розглядає обчислення як моделювання поведінки об’єктів, які представлені обчислювальними абстракціями. Припустимо, наша мета поліпшити навички гри в покер, тому слід навчитися обчислювати ймовірність випадіння різних комбінацій гральних карт.

Для цього треба описати (сформувати абстрактну модель) наступне:

- поняття масть;
- поняття гральна карта;
- процедуру перемішування (перетасовки) гральних карт карткової колоди;
- правила за якими оперують мастями та зростом карт.

Можливо використовувати назви мастей: піки, черви, бубни і трефи, але технічно масті можно представити як цілими числами. Це внутрішнє уявлення приватно і не впливає на розрахунки. Зрозуміло, що реальні карткові колоди мають фактично різні фізичні властивості, але очікувана поведінка при оперуванні з ними є однаковою.

Надалі будемо застосовувати термін *абстрактний тип даних, АТД* (abstract data type, ADT) для позначення того, що визначається користувачем для розширення «вбудованих» типів даних мови програмування. АТД складається з набору можливих значень для відповідних характеристик та операцій, які можуть оперують ними або впливати на ці значення. Наприклад, в мові С немає такого типу даних для комплексне число, а мова С++ дозволяє додати такий тип і інтегрувати його з існуючими (вбудованими типами даних).

Об’єкти (objects) це екземпляри класу. ООП дозволяє оперувати АТД. ООП використовує механізм *успадкування* (inheritance) для породження похідних типів даних із типів даних, які сформував розробник програмного забезпечення.

У ООП об’єкти відповідають за свою поведінку. Наприклад, многочлени, комплексні числа, цілі числа, числа з плаваючою точкою - все це об’єкти, які «розуміють» операцію складання.

Кожен тип має пов'язаний з ним програмний код (алгоритм дій) для виконання операції складання. Компілятор надає належний код для цілих і чисел з плаваючою точкою.

Абстракний тип даних «многочлен» містить функцію, що визначає складання, відповідно до особливостей своєї реалізації. Постачальник (розробник) АТД повинен включити до нього код, який реалізує будь-яку поведінку, яку зазвичай можлово очікувати від відповідних об'єктів. Факт, що об'єкт самостійно «відповідає» за власну «поведінку», значно спрощує завдання програмування для користувача цього об'єкта.

Уявімо собі клас об'єктів під назвою «фігури». Якщо ми хочемо намалювати на екрані якусь фігуру, нам треба знати, де буде знаходитися її центр і як її малювати. Деякі фігури, наприклад, багатокутники, відносно легко намалювати. У загальному випадку, процедура малювання фігури може бути трудомістким, так як, можливо, буде потрібно зберігати велику кількість окремих граничних точок. Навпаки, варіант з багатокутником безсумнівно зручний. Якщо окрема фігура прекрасно розуміє, як себе намалювати, програміст при використанні такої фігури повинен лише передати об'єкту повідомлення «намалювати (ся)».

У C++ нове поняття класів надає механізм *інкапсуляції* (encapsulation) для реалізації АТД. Інкапсуляція поєднує в собі, з одного боку, внутрішні деталі реалізації конкретного типу і, з іншого, доступні ззовні операції і функції, які можуть діяти на об'єкти цього типу. Деталі реалізації можуть бути недоступні для програми, яка використовує даний тип. Наприклад, стек може бути реалізований як масив фіксованої довжини, а доступні операції повинні включати в себе функції `push` (покласти у стек) і `pop` (витягти зі стеку). Зміна внутрішньої реалізації на зв'язний список не повинно вплинути на те, як `push` і `pop` використовуються зовні класу. Код, який використовує АТД, називається *клієнтом АТД*. Реалізація стека прихована від його клієнтів.

Концепція ООП оперує поняттями:

- моделювання дій з реального світу;
- типи даних, які розроблені користувачем;
- закриття деталей реалізації (інкапсуляція);
- багаторазове використання програмного коду (успадкування);
- інтерпретація викликів функцій на етапі виконання (поліморфізм).

Деякі поняття достатньо розпливчасті, деякі - абстрактні, інші мають узагальний характер.

Приклад програми мовою С++

Програмування мовою С++ це поєднання програмування на низькому (системному) та високому (проблемному) рівнях. Мова С була розроблена як системна мова, близька до машинної мови. С++ доповнена об'єктно-орієнтованими властивостями, які дозволяють програмісту створювати або імпортувати бібліотеки, притаманні конкретному завданню. Користувач може писати код на проблемному рівні, до того йому доступний машиний рівень для реалізації деталей.

Приклад 1. Привітання (вивід інформації)

```
// Вітаємось зі світом на C++
#include <iostream.h>    // бібліотека вводу-виводу
#include <string>        // строковий тип

using namespace std;     // простір імен ст. бібліотек

inline void pr_message(string s = "Hello world!")
{
     cout << s << endl;
}

int main()
{
     pr_message();
}
```

При виконанні програма надрукує:

Hello world!

Програма мовою C++ це послідовність оголошень і функцій. Виконання програми починається з функції `main()`. Коли програма компілюється, спочатку виконуються директиви препроцесора, наприклад, директива #include, яка імпортує зазначений файл, зазвичай визначення бібліотеки. У прикладі, бібліотека функцій вводу-виводу описана файлах `iostream` або `iostream.h`.

Бібліотека `string` є частиною стандартної бібліотеки C++ і повинна бути включена для використання оголошення `string`.

«Простір імен» `std` зарезервовано для використання оголошень зі стандартних бібліотек. Термін «простір імен» був доданий до стандарту ANSI мови програмування C++ з метою надання і розширення «області видимості». Це дозволяє різним постачальникам програмного коду уникати конфліктів з глобальними іменами.

Оголошення `using` дозволяє використовувати ідентифікатори з стандартної бібліотеки без введення повного імені ідентифікатора. Відсутності оголошення `using` зобов'язує програміста написати повне ім'я, у прикладі був би використуваний запис `std::string`.

Символ `//` (подвійна коса риска) використовують для додавання коментарів на кінці рядка. Текст програми може розташовуватися в будь-якому місці сторінки. Пропуски рядків, прогалини, коментарі, абзацний відступи використовують для написання добре документованої програми. На семантику (зміст) програми він і не впливає.

Модифікатор `inline` функції `pr_message()` повідомляє компілятору, що у разі можливості можна відмовитися від генерації інструкцій виклику і повернення, що дозволяє геререувати більш ефективний код програми.

З тексту зрозуміло, що функція `pr_message()` отримує параметр `s`, тип якого – строковий, а значенням за замовчуванням `"Hello world!"`. Це

означає, що коли виконується вклик функцій із порожнім список параметрів, виконується `pr_message("Hello world!")`.

Ідентифікатор `cout` визначено у файлі `iostream` як стандартний вихідний потік. В більшості систем програмування стандартний вихідний поток пов'язаний із виводом інформації на екран. Ідентифікатор `endl` це стандартний маніпулятор (manipulator), який очищає вихідний буфер і після друку усього вмісту буфера і здійснює перехід на новий рядок. Оператор `<<` направляє в `cout` все, що напсано після нього.

У C++ функція може повертати значення невизначеного типу, або нічого не повератати до функції яка її викликала. У такому випадку використовують ключеве слово `void`. Це означає, що функція не повертає ніякого значення, як у випадку з `pr_message()`.

Функції `main()` повертає до системи, що її викликала, ціле значення, зазвичай - значення 0 «Нуль». Це ознака нормального завершення програми. Інші значення `main()` повертає явно, за допомогою інструкції `return`.

Ось ще один варіант `main()`:

Приклад 2. Параметри функції

```
int main()
{
     pr_message();
     pr_message("Остап Бендер");
     pr_message("Пора обідати!");
}
```

При виконанні програма надрукує:

```
Hello world!
Остап Бендер
Пора обідати!
```

Інкапсуляція і розширюваність типів

ООП це ідеологія і технологія створення програмного забезпечення, за якою «дані» і «поведінка» «упаковані» разом. Таку «упаковку» називають *інкапсуляція* («у капсулі»).

Інкапсуляція дозволяє розробнику створювати нові типи даних, які розширюють власні (вбудовані) типи даних мови програмування і реалізують взаємодію з ними.

Розширення типів це можливість доповнювати мову користувальницькими типами даних, якими можна оперувати так саме як і вбудованими типами даних мови програмування.

Абстрактний тип даних (АТД) це опис поведінки ідеального типу. Наприклад, для типу даних «текстовий рядок» (далі рядок) операції конкатенація або друк реалізують певну поведінку. Такі операції (функції, процедури) називають *методами.*

Реалізація АТД (програмний код) може мати обмеження. Наприклад, рядки можуть бути обмежені довжиною (кількістю текстових символів - елементів). Обмеження впливають на поведінку, яка відкрита на зовні (доступна) для інших компонентів програми. У той час як внутрішні або закриті деталі реалізації не впливають безпосердньо на те, як користувач бачить об’єкт. Наприклад, рядок іноді реалізується як масив текстових символів. При цьому внутрішня базова адреса елементів масиву і його ім’я є несуттєвими для користувача.

На термінологію ООП мала значний вплив мова програмування Smalltalk. Тут були започатковані поняття *повідомлення і метод,* які зайняли місце традиційних понять *виклик функції і функція-член.*

Інкапсуляція це засоби приховування внутрішніх деталей при наданні інтерфейсу до абстрактного типу даних. Для забезпечення інкапсуляції використовують оголошення класу (`class`) і структури (`struct`) у поєднанні з ключовими словами доступу `private` (закритий), `protected` (захищений) і `public` (відкритий).

Ключове слово public вказує, що доступ до членів відкритий усім без обмежень. За замовчуванням члени класу закриті, тобто доступ до їх вмісту мають лише методи класу. Відкриті члени доступні для будь-якої функції всередині області видимості оголошення класу. Обмеження доступу (приховування даних) дозволяє закрити частину реалізації класу і тим запобігати непередбаченим змінам даних.

Далі дамо опис класу із назвою my_string, який реалізує обмежену форму поняття «сукупніть друкованих символів» - символьна строка (рядок).

Приклад 3. Опис класу

```
// Найпростіша реалізація типу    my_string
const int max_len = 255;
class my_string
{
     public:        // загальний доступ до інтерфейсу
     void assign(const char* st);

     int length() const
     {
     return len;
     }

     void print() const
     {
     cout << s << "\nДовжина:" << len << endl;
     }

     private:       // обмежений доступ до реалізації
     char s[max_len];
     int len;
};
```

Закритою частиною класу є:

- масив із кількістю символів max_len ;
- змінна len, у якій зберігається довжина рядка.

Оголошення функцій-членів дозволяє абстрактному типу даних мати окремі функції, що оперують (можливо змінюють значениия) елементами із «закритої» частини.

Наприклад, функція-член `length()` повертає довжину рядка. Функція-член `print()` виводить рядок і його довжину. Функція-член `assign()` записує «символьний рядок» у «закриту» змінну `s,` обчислює довжину рядка, яку зберігає у «закритій» змінній `len.`

Методи, які не змінюють значення змінних, оголошені із модіфікатором `const.`

Після сформованного опису можна використовувати клас `my_string` у програмі так само як вбудований (власний) тип даних мови програмування.

Код, який використовує абстрактний тип даних, називають *клієнтом* типу. Для маніпуляції зі змінними типу `my_string` (об'єктами `my_string`) будуть використані тільки «відкриті» методи (методи, які оголошені в області `public`).

Приклад 4. Використання класу

```
// Перевірка класу my_string
int main ()
{
     my_string one;
     my_string two;
     char three[40] = {"Мене звуть Чарльз Беббідж."};

     one.assign("Мене звуть Алан Тьюринг.");
     two.assign(three);

     cout << three;
     cout << "\nДовжина:" << strlen (three) << endl;

     // Друк найкоротшої змінної з one і two
     if ( one.length() <= two.length() )
          one.print();
     else
          two.print ();
}
```

Змінні `one` і `two` мають тип `my_string`.

Змінна `three` має модифікатор типу – «покажчик» на символ. Вона несумісна з типом даних `my_string` і тому застосувати методи класу (типу даних `my_string`) до неї НЕМОЖЛИВО.

Функції-члени викликаються з використанням *оператора "крапка" (оператор доступу до члена класу/структури).* Із визначень зрозуміло, що функції-члени оперують зі значеннями «закритих атрибутів» (даними відповідних змінних). Результат роботи програми тут:

```
Мене звуть Чарльз Беббідж.
Довжина: 26
Мене звуть Алан Тьюринг.
Довжина: 24
```

Конструктори і деструктори

У термінології ООП змінна називається *об'єктом.* Функцію-член, яка реалізує функції ініціалізації об'єкта класу, називають *конструктор* (constructor). У багатьох випадках ініціалізація передбачає динамічний розподіл пам'яті. Конструктор викликається щоразу, коли створюється об'єкт класу. Конструктор з одним аргументом може виконувати перетворення типів, якщо у його оголошенні не використовується ключове слово `explicit` (явний).

Деструктор (destructor) це функція-член, яка «знищує об'єкт» класу і вивільняє ресурси. Цей процес часто передбачає динамічне звільнення пам'яті. Деструктор викликається неявно, коли автоматичний об'єкт виходить за межі області видимості.

Додамо дещо до опису класу `my_string`. Змінемо спосіб надання ресурсу (виділення пам'яті) для атрибуту - «закритої» змінної `s`. Попередній варіант – пам'ять надавалася СТАТИЧНО, новий варіант – надання пам'яті динамічно.

Закритий член даних (масив) або «покажчик». Оновлений клас буде використовувати конструктор для динамічного виділення пам’яті. Для цього застосуємо оператор new.

Приклад 5. Конструктори і деструктори

```
// Реалізація динамічної my_string
class my_string
{
public:        // конструктор

explicit my_string (int n)
{
     s = new char[n+1];
     len = n;
}

void assign(const char* st);

int length() const {return len; }

void print() const
{
     cout << s << "\ nДліна:" << len << endl;
}

     private:
     char *s;
     int len;
};
```

Для такої реалізації стане у нагоді варіант функції-члена assign(), яка виділяє пам’ять динамічно (за потребою):

```
void my_string::assign(const char* str)
{
     delete [] s;
     len = strlen(str);
     s = new char [len + 1];
     strcpy (s, str);
}
```

Ім’я конструктора збігається з ім’ям класу. При виконання операції резервування пам’яті і у подальшому для ініціалізації об’єктів, конструктор використовує оператор new. Це унарний оператор, який у якості аргументу отримує тип даних, зокрема, розмір масиву певного типу. Оператор `new` виділяє необхідний обсяг пам’яті для зберігання даних відповідного типу і повертає покажчик з адресою зарезервованої пам’яті. У попередньому прикладі у оперативній пам’яті має бути зарезервовано і надано `n+1` байт. Так, після оголошення:

```
my_string a(40), b(100);
```

Для змінної `a` потрібно 41 байт і 101 байт буде виділено для змінної `b`. Доданий один байт для символу кінця «символьної строки» (рядка) так званий «нуль-символ» (`'\0'`).

Оператор `new` виділяє пам’ять на постійній основі, і вона не звільняється автоматично при виході з області видимості. Якщо потрібно звільнити пам’ять, до опису класу слід додати і реалізовати (написати програмний код) спеціальну функцію-член - деструктор. Деструктор є звичайною функцією-членом, ім’я якої збігається з іменем класу, але його починає символ `'~'` (тильда). Для «знищення» об’єкту, ресурси (пам’ять) якому зарезервовані оператором `new`, деструктор використовує унарний оператор `delete` - щоб автоматично звільнити пам’ять, на яку посилається відповідний вираз-покажчик.

```
// Додано як функція-член до класу my_string
~My_string() {delete []s;} // деструктор
```

Поширеної практикою є перевантаження конструктора. Техніка , для формування кількох функцій-конструкторів. Це дозволяє визначити декілька способів ініціалізації об’єкта. Розглянемо, наприклад, ініціалізацію `my_string` покажчиком на символьне значення. Конструктор буде виглядати так:

```
my_string (const char* p)
{
     len = strlen(p);
     s = new char[len + 1];
     strcpy (s, p);
}
```

Типове оголошення, що викликає таку версію конструктора:

```
char* str = "Я прийшов пішки.";
my_string a("Я приїхав автобусом.");
my_string b(str);
```

Бажано також мати конструктор без аргументів:

```
my_string() {
     len = 0;
     s = new char[1];
}
```

Тут оголошення проводиться без аргументів, і за замовчуванням буде виділений один байт пам’яті. Наступна послідовність інструкцій програми викличе усе три варіанти конструктора об’єктів класу:

```
my_string  a;
my_string  b(20);
my_string  c("Я приїхав верхи.");
```

Перевантажений конструктор обирається залежно від форми оголошення.

Змінна a не має параметрів, тому під неї буде виділено один байт.

У змінної b є цілий параметр, і для неї відведено 21 байт.

Змінна c конструюється за допомогою параметра – «покажчик на символьний рядок» ("Я приїхав верхи."). Для неї виділено 18 байт, причому аргумент функції – «символьний рядок» копіюється в закритий член s.

Перевантаження

Перевантаженням (overloading) називається практика надання декількох значень оператору або функції. Вибір конкретного значення залежить від типів аргументів, які отримують оператори або функції.

Наведемо приклад перевантаження функції `print()` з попереднього прикладу. Це буде друге визначення функції `print()`.

Приклад 6. Перевантаження функцій

```
class my_string
{
     public: // загальний доступ

void print() const
{
     cout << s << "\nДовжина:" << len << endl;
}

void print(int n) const
{
     for(int i = 0; i<n; ++i)
     cout << s << endl ;
}

}
```

Нова версія функції `print()` отримує у якості аргументу значення «цілого» типу. Ця версія функції виведе «символний рядок» `n` разів.

```
three.print(2); // друк рядка з об'єкту three двічі
three.print(-1); // рядок з об'єкту three не друкувати
```

Функції-члени – operator. Функції «друзі»

Більшість операторів C++ може бути перевантажено. Наприклад, перевантажимо операцію `+` (додавання) для оформлення операції конкатенації рядків. Нам знадобляться нові поняття, з якими пов'язані ключові слова: `friend` і `operator`.

Ключове слово `operator` розташовується перед знаком операції і замінює ім'я функції.

Ключове слово `friend` надає функції доступ до закритих членів класу. «Дружня» (`friend`) функція не є членом класу, але має привілеї функції-члена класу, «другом» якого вона оголошена.

Приклад 7. Перевантаження операцій

```
// перевантажена операція +
class my_string
{
public:
my_string()
{
     len = 0;
     s = new char[1];
}

explicit my_string(int n)
{
     s = new char[n + 1];
     len = n;
}

void assign(const char *st);

int length() const
{
     return len;
}

void print() const
{
     cout << s << "\nДовжина:" << len << endl;
}

my_string& operator=(const my_string &a);

friend my_string& operator+(const my_string& a, const my_string&
b);

private:
     char *s;
```

```
     int len;
};

// Перевантаження операції додавання ‘+’
my_string& operator+(const my_string& a, const my_string& b)
{
     my_string *temp = new my_string( a.len + b.len);
     strcpy(temp->s, a.s);
     strcat(temp->s, b.s);
     return *temp;
}

void print(const char *c)
// (не плутати з print з my_string !)
{
cout << з << "\ n Довжина:" << strlen ( c ) << endl ;
}

int main()
{
     my_string one, two, both;
     char three[40] = {"Мене звуть Чарльз Беббідж."};
     one.assign(«Мене звуть Алан Тьюринг.");
     two.assign(three);
     print(three);
// Друк наібільш короткого рядки з one або two
     if (one.length() <= two.length() )
          one.print(); // виклик функції-члена print
     else
          two.print ();

     both = one + two; // конкатенація
     both.print();
}
```

Розбір функції `operator+()`

```
my_string& operator+(const my_string &a, const my_string &b)
```

«Плюс» перевантажений. Операция має два аргументи, обидва - `my_string`. Аргументи передаються по посиланню. Запис виду *тип*

&ідентифікатор оголошує ідентифікатор як змінну-посилання. Ключове слово `const` показує, що аргументи не можуть бути змінені.

```
my_string * temp = new my_string( a.len + b.len );
```

Функція повинна повернути значення типу `my_string`. Цей локальний покажчик буде використаний, щоб повернути значення `my_string` після виконання конкатенації. Для виділення обсягу пам'яті використовується оператор `new` .

```
return *temp;
```

Операція «отримання адреси покажчика» `temp` повертає адресу об'єднаної строки `my_string`.

Шаблони і узагальнене програмування

Ключове слово `template` є інструментом реалізації наступної важливої концепції об'єктно-орієнтованого програмування - *параметричний поліморфізм.*

Параметричний поліморфізм дозволяє використовувати єдиний програмний код алгоритму для різних типів даних. При цьому тип даних є параметром коду. Така технологія є ефективною при описі загальних *контейнерних класів.*

Контейнерні класи використовують для зберігання даних. Такими класами є:

- `стек` (англ. `Stack` - стопка; читається *стек*) - абстрактний тип даних, являє собою список елементів, які організовані за правилом обслуговування `LIFO` (англ. `Last In - First Out` , «останнім прийшов - першим вийшов»);

- вектор - одновимірний масив (vector) шаблон зі стандартної бібліотеки C++, що реалізує динамічний масив (контейнер std::vector);
- контейнер «дерево»;
- контейнер «Список».

Приклад 8. Контейнери. Клас Stack як параметризований тип

```
// Реалізація шаблону stack
template <class TYPE>
class stack
{

public:
explicit stack (int size = 1000): max_len(size)
{
     s=new TYPE [size];
     top = EMPTY;
}

~stack()
{
     delete []s;
}

void reset()        // очистити стек
{
     top = EMPTY;
}

void push(TYPE c)   // покласти у стек
{
     s[++top] = c;
}

TYPE pop()               // витягти зі стеку
{
     return s[top--];
}
TYPE top_of()       // вважати верхній елемент
{
     return s [top];
```

```
}

bool empty()   // перевірка - стек порожній?
{
     return (top == EMPTY);
}

bool full()         // перевірка - стек заповнений?
{
     return (top == max_len-1);
}

private:
     enum {EMPTY = -1);
     TYPE * s;
     int max_len ;  // максимальна довжина
     int top;       // вершина
};
```

Оголошення класу виглядає так:

```
template <class 'ідентифікатор'>
```

Ідентифікатор - це аргумент шаблону. Із природи аргументу зрозуміло, що на його місце можна записати будь-яке значениия. У наведеному випадку аргумент це ім'я класу (читай – назву типу даних, розумій – ім'я будь-якого абстрактного типу даних).

Скрізь у програмному коді для визначення класу аргумент шаблону використовується у якості імені типу.

Значення аргументу визначає розробник при формуванні програми для конкретного алгоритму, іншими словами при фактичному оголошенні.

Приклад 9. Оголошення стеку для різних типів даних

```
stack <char>stk_ch ; // стек з 1000 символів
stack <char *>stk_str(200); // стек з 200 покажчиків
stack <complex>stk_cmplx(100); // стек із 100 компл. чисел
```

Механізм шаблонів дозволяє не переписувати оголошення класів, які відрізняють лише типом даних.

При роботі із шаблонами, у оголошенні завжди треба використовувати кутові дужки <>, для визначення конкретного типу даних який буде оброблятися за шаблоном

Приклад 10. Шаблон Stack для роботи із текстовими рядками

```
// Звернення послідовності покажчиків char*,
// представляють рядок
void reverse( char* str[], int n)
{
     stack <char*>stk(n); // елементи стеку char*

     for(int i = 0; i<n; ++i)
          stk.push(str[i]);
     for(int i = 0; i<n; ++i)
          str[i] = stk.pop();
}
```

Функція reverse() використовує стек stack <char*>, який приймає n рядків, а потім рядки витягають в зворотному порядку.

Приклад 11. Шаблон Stack для роботи із комплексними числами

```
// ініціалізація стеку комплексними числами з масиву
void init(complex c[], stack<complex>&stk, n)
{
     for(int i = 0; i<n; ++i)
          stk.push(c[i]);
}
```

У функції init() змінна типу stack<complex> передається за посилання (за адресою), у стек «кладуть» n комплексних чисел.

Стандартна бібліотека шаблонів (STL)

Стандартна бібліотека шаблонів (Standard Template Library - STL) дозволяє використовувати узагальнене програмування для поширених структур даних і алгоритмів.

Бібліотека вміщує конструкції мови програмування, які у сукупності описують:

- контейнерні класи (вектори, черги, відображення);
- засоби перебору елементів контейнера (сукупності) за допомогою класів ітераторів;
- алгоритми для обслуговування контейнерів - функції сортування і пошуку даних і т. і.

Розглянемо контейнери, ітератори і алгоритми, які створюють основу для узагальненого програмування.

Бібліотека містить опис шаблонів. Структура бібліотеки цілком ортогональна – іншими словами усі компоненти є незалежними один від одного. Тому компоненти бібліотеки можна комбінувати один з одним.

Параметрами компонентів можуть виступати як власні (вбудовані) типи даних мови програмування, так типи, які спроектовані (визначені) користувачем. Інакше - будь-які абстрактні типи даних.

Приклад 12. STL. контейнер List (список)

```
// Використання контейнера list (список)
#include <iostream>
#include <list> // контейнер списків
#include <numeric> // для функції accumulate ()

using namespace std ;

void print(const list<double>& lst )
{// використовуємо итератор для подорожі по 1 st
list <double>::const_iterator where;
for(where=lst.begin(); where != lst.end(); ++where)
     cout << *where << '\t';
cout << endl;
}
```

```
int main()
{
double w[4] = {0.9, 0.8,  88, -99.99};
list <double>z;
for (int i = 0; i <4; ++i)
     z.push_front(w[i]);

print(z);
z.sort();
print(z);

cout << "Сума дорівнює ";
cout << accumulate(z.begin(), z.end(), 0.0) << endl;
}
```

У прикладі контейнер List використаний для зберігання значень – число із подвійною точністю.

Елементи масиву по одному «кладемо» до списка.

Функція `print()` використовує ітератор для друку по черзі елементів списку.

Ітератори мають стандартний інтерфейс, частиною якого є функції-члени `begin()` і `end()` для визначення початку і кінця контейнера.

Крім того, інтерфейс включає в себе алгоритм сортування - функцію-член `sort()`.

Функція `accumulate()` використовує `0.0` якості початкового значення та обчислює суму елементів контейнера шляхом проходу від початкової позиції `z.begin()` до кінцевої позиції `z.end()`.

Спадкування

ООП надає розробникам механізм спадкування, що дозволяє повторно використати програмний код. Новий клас (абстрактний тип даних) як правило розробляють спираючись на попередні розробки, структури, типи даних. Тому він може отримати не тільки властивості (атрибути), але і поведінку (методи), яка схожа на попередників. Це виявляє їх зв’язок, який ми називаемо «спадковістю».

З таким міркувань новий клас отримує характер похідного або спадкового класу, а його попередник виступає у статусі базового або батьківського класу.

Похідний клас спадковує опис базового класу, використовує члени базового класу, може змінювати і доповнювати їх.

Відносини спадковості є ієрархічними.

Ієрархія - це метод розробки, що дозволяє копіювати елементи в усьому їх різноманітті і складності. Вона вводить класифікацію об’єктів.

Наприклад, у періодичній системі хімічних елементів Д. І. Мєнделєєва є гази.

Вони мають властивості, притаманні усім елементам системи.

Інертні гази є важливим підкласом класу гази.

Ієрархія полягає у тому, що інертний газ, наприклад, аргон - це газ, а газ, в свою чергу є елементом системи. Така ієрархія дозволяє легко тлумачити поведінку інертних газів.

Відомо, що атоми газу містять протони та електрони. Це також вірно і для інших елементів системи.

Відомо, що гази перебувають у газоподібному стані при кімнатній температурі, як усі гази.

Відомо, що жоден газ з підкласу інертних газів не вступає в звичайну хімічну реакцію з іншими елементами системи, і це властивість всіх інертних газів.

Об’єктно-орієнтоване проектування

Вибери належну сукупність типів.

Спроектуй взаємозв’язку типів в коді, використовуючи успадкування.

Розглянемо наступну задачу.

Візмемо до розробки структуру і програмне забезпечення для бази даних «Облік контингенту студентів вищого навчального закладу (університету)».

Студентський відділ повинен вести облік студентів різних категорій. Базовий клас, який необхідно розробити, буде фіксувати опис студента. Основними категоріями учнів будемо вважати дві - це аспірант і студент (НЕ аспірант).

Приклад 13. Спадкування

```
// тип фінансової підтримки
enum support { ta, ra, fellowship, other };

// курс (рік навчання )
enum year {fresh, soph, junior, senior, grad};

class student // клас студентів
{
public:
// конструктору передаються ім’я
// номер , середній бал , курс
student(char* nm, int id, double g, year x);
void print() const;
private:
     int student_id; // номер
     double gpa; // середній бал
     year y; // курс
     char name[30]; // ім’я
}

class grad_student: public student // клас аспірантів
{
public:
// конструктору додатково передаються:
// тип фінансової підтримки,
// назва кафедри, тема дисертації
grad_student(char* nm, int id, double g, year x, support t, char*
d, char* th);
void print() const;
private:
     support s; //фінансова підтримка
     char dept[10]; // кафедра
     char thesis[80]; // назва дисертації
};
```

У прикладі клас `grad_student` є похідним класом від базового класу `student`.

Використання у заголовку похідного класу ключового слова `public` означає, що відкриті члени класу `student` повинні спадковуватися як відкриті члени `grad_student`.

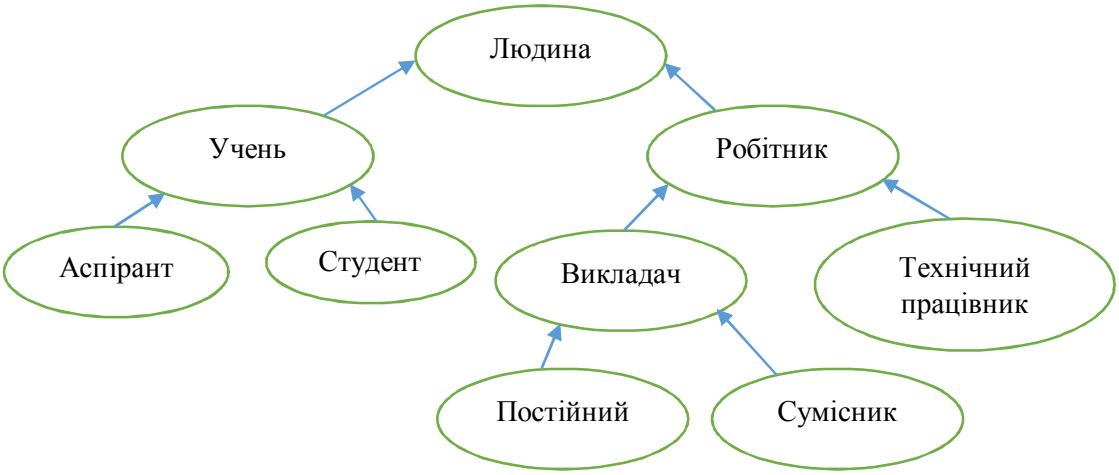
Закриті члени базового класу недоступні похідному класу.

Відкрите спадкування означає також, що похідний клас `grad_student` є підтипом `student` .

Структури спадкування утворюють каркас для побудови доволі загальних систем.

Наприклад, база даних, яка містить інформацію про усіх людей в університеті, може бути спадкована від базового класу `person` (людина). Базовий клас `student` можна використовувати для створення похідного класу студентів-юристів, як наступною значущою категорії об'єктів.

Аналогічно, `person` може стати базовим класом для різних категорій



працівників. Ієрархічна структура зображена на малюнку.

Поліморфізм

`Поліморфна` функція/операція має кілька варіантів реалізації. Наприклад, операція ділення поліморфа. Якщо аргументи операції цілі числа,

то виконується цілочисельне ділення. Якщо хоча б один з аргументів має тип число з плаваючою точкою, використовується розподіл для чисел з плаваючою точкою.

Приклад 14. Поліморфізм, операція ділення

```
 a / b // тип визначається правилами приведення
```

Приклад 15. Поліморфізм, операція виводу

```
cout << a; // поліморфізм через перевантажену функцію
```

Операція зсуву '<<' реалізована як функція, яка вміє виводити об'єкти типу a. Якщо a - ціле, виконується відображення на екрані цілого оцифровуваного значення, якщо a - з плаваючою точкою, то виконується відображення на екрані значення із плаваючою точкою.

Ім'я функції або оператора може бути перевантажено. Проте яка з функцій буде викликана під час виконання програми, визначається *сигнатурою функції,* тобто типами аргументів у списку параметрів функції.

Поліморфізм локалізує відповідальність за поведінку. Клієнтський код часто не вимагає перегляду, коли до системи додається функціональність за допомогою покращення коду абстрактного типу даних.

Реалізація набору процедур для завдання типу геометричної фігури може грунтуватися на вичерпному описі довільної фігури.

Приклад 16. Поліморфізм, об'єкт «геометрична фігура»

```
srtuct shape // фігура - коло, прямокутник
{
enum { CIRCLE, RECTANGLE, } e_val;
double center; // центр
double radius; // радіус
};
```

До опису долучені всі атрибути (члени) довільної геометричної фігури.

По-перше, змінну для ідентифікації, це змінна типу enum. За такої об'яви функція «Обчислення площі пласкої фігури» буде виглядати так.

Приклад 17. Поліморфізм, функція «обчислення площі фігури»

```
double area(shape* s)
{
     switch (s->e_val)
     {
     case CIRCLE: return (PI * s->radius * s->radius);
     case RECTANGLE: return (s->heght * s->width);
     }
}
```

Бажано розширити список геометричних фігур.

Це означає оновлення програмного коду для додавання роботи з новою фігурою.

Знадобиться додати інструкції до оператора `switch` рядок `case` в тілі коду і додаткові члени структури.

Це спричинить зміни в тілі коду, оскільки кожна процедура побудована так, що доведеться додава `case`, навіть коли він відіграє роль додаткової мітки у існуючому `case`.

Таким чином, локальне розширення вимагає глобальних змін.

Об’єктно-орієнтована технологія програмування спирається на шдею ієрархії геометричних фігур для вирішення такої задачі. Ієрархія очевидна.

Існує загальне поняття фігура - клас «Геометрична фігура». Коло і прямокутник успадковуються з фігури. У процесі оновлення програмного коду можливі доповнення виконаємо в новому похідному класі. Таким чином додаткові описи локалізовані.

Розробник заміщає сенс будь-якої зміни процедури.

У прикладі, треба додати ще один вираз (формулу) для обчислення площі.

Клієнтський код, який не використовує новий тип, залишається без змін. Код, який розширений новим типом , змінюється мінімально.

Програма, яка написана за викладеної схемою, використовує `shape` як *абстрактний базовий клас.* Це клас, який містить хоча б одну чисто віртуальну функцію .

Приклад 18. Поліморфізм, оновлення класу (віртуальна функція)

```
// shape - абстрактний базовий клас (фігура)
class shape
{
public:

virtual double area() = 0; // віртуальна функція - обчислити площу
фігури
};

// клас прямокутник – похідний клас
class rectangle: public shape
{
public:
rectangle (double h, double w): height(h), width(w){}
double area()
{
     return (height * width);
}
     private:
     double height; // висота
     double width; // ширина
};

// коло
class circle: public shape
{
public:
circle(double r): radius(r)  {}
double area()
{
     return (3.14159 * radius * radius);
}
     private:
     double radius; // радіус
};
```

Клієнтський код для обчислення довільної площі поліморфний. Належна функція `area()` вибирається на етапі виконання:

```
shape * ptr_shape;
cout << "Площа =" << ptr_shape->area();
```

Додамо до списку фігур – квадрат. Доповнимо ієрархію типів і сформуємо опис класу `square` (квадрат):

```
class square: public rectangle
{
public:
square (double h): rectangle(h, h) {}
double area()
{
     return (rectangle:: area());
}
};
```

Клієнтський код залишається без змін, на відміну від не коду, який записаний без урахування ідеології і принципів об'єктно-орієнтованого програмування.

Ієрархічна схема повинна бути спрямована на мінімізацію інтерфейса передачі параметрів.

Кожен рівень ієрархії спрямований на `інкапсуляцію` («приховання в собі» в межах своєї реалізації) деталей побудови, на які впливає виклик функції.

У технології програмування, яка відмінна від ООП, такі зміни іноді можуть супроводжуватися внесенням змін до глобальних даних і умов.

Така практика вразлива, бо веде програмування з побічними ефектами, що ускладнює налагодження, оновлення і підтримку коду.

Обробка виключень у C++

C++ вводить механізм обробки виключень, чутливий до контексту. Контекстом для збудження виключення є блок `try`. Обробники, які оголошуються ключовим словом `catch`, знаходяться нижче блоку `try`.

Виняток збуджується за допомогою ключового слова `throw`. Виняток буде оброблено викликом відповідного обробника, обраного зі списку, який йде відразу за блоком `try`.

Приклад 19. Обробка виключень (інструкції try, throw, catch)

```
// Конструктор стека з винятками
stack::stack(int n)
{
     if(n <1)
     throw(n); // очікуємо позитивне значення
     p = new char[n]; // створюється стек із символами
     if( p == 0) // new повертає 0 при невдачі
     throw(«ВІЛЬНА ПАМ’ЯТЬ вичерпані");

     top = EMPTY;
     max_len = n;
}

void g()
{
     try{
          stack a(n);
          stack b(n);
     }
catch(int n) { } // некоректний розмір
catch( char* error) { } // вільна пам’ять вичерпана
}
```

Перший throw() має цілий аргумент і відповідає сигнатурі catch(int n). Цей обробник повинен виконати дії, коли до конструктора у якості аргументу передається некоректний розмір масиву. Наприклад, вивести повідомлення про помилку і перервати програму.

Другий throw() отримує в якості аргументу покажчик на символ і відповідає сигнатурі catch(char *error) .

Переваги об’єктно-орієнтованого програмування

Центральним елементом ООП є інкапсуляція сукупності даних і операцій. Поняття класу з його функціями-членами і членами-даними надає інструмент для реалізації інкапсуляції. Змінні класу є об’єктами, якими можна управляти.

Класи забезпечують приховування даних. Можливо дозволити або обмежити доступ до деталей реалізації чи будь-якої групи функцій, що забезпечує модульність і надійність.

Важливою концепцією, на яку цілеспрямоване ООП є повторне використання програмного коду за допомогою механізму успадкування. Похідний (новий) клас створюють спираючись абстракції і попередні розробки у формі базових (існуючих) класів. При створенні похідного класу базовий клас можливо не тільки доповнити але і змінити (перевантажити) його методи. Таким чином створюють ієрархії родинних типів даних, які використовують загальний код.

Об’єктно-орієнтоване програмування глибше, ніж процедурне програмування. При розробці програмного забезпечення, до алгоритмізації та кодування додається розробка типів даних предметної області.

Використання ООП дає:

модульність программного забезпечення, отже, програми більш зрозумілі і прості для модифікації і обслуговування;

повторне використання програмного коду.

ООП = розширюваність типів + поліморфізм .

Список використаних джерел

1. Авдеев В. А. Периферийные устройства. Интерфейсы, схемотехника, программирование / В. А. Авдеев. – Москва : ДМК Пресс, 2009 – ISBN: 978-5-94074-505-1.
2. Адельсон-Вельский Г. М Программирование игр /Г. М. Адельсон-Вельский, В. Л. Арлазаров, М. В. Донской. – Москва : Наука, 1978. – 256 с. : ил.
3. Бондарев В. М. Основы программирования / В. М. Бондарев, В. И. Рублинецкий, Е. Г. Качко. – Харьков : Фолио ; Ростов-на-Дону : Феникс, 1998. – 368 с. : ил. – ISBN 966-03-0313-0.
4. Буч Г. Язык UML : руководство пользователя / Г. Буч, Рамбо, А. Джекобсон ; пер. с англ. – Москва : ДМК, 2000. – 432 с. : ил. – (Серия для программистов). – ISBN 5-93700-009-9.
5. Бьерн Страуструп Язык программирования C++ / Бьерн Страуструп. – [Б. м.] : Бином, 2011.
6. Вайнер Р. C++ изнутри / Р. Вайнер, Л. Пинсон ;пер. с англ. – Киев : ДиаСофт, 1993. – 304 с. : ил. – ISBN 5-87554-079.
7. Гарбер М. Введение в SQL / Мартин Гарбер ; пер. с англ. – Москва : Лори, 1996.
8. Гарднер М. Крестики-нолики / М. Гарднер ; пер.с англ. – Москва : Мир, 1988. – 352 с. ил. ISBN 5-03-001234-6.
9. Дунаев В. В. Самоучитель JavaScript / В. В. Дунаев. – Санкт-Петербург : Питер, 2003. – 395 с. : ил. – ISBN 5-94723-533-1.
10. Дьюхарст С. Программирование на C++ /С. Дьюхарст, К. Старк ; пер. с англ. – Киев : ДиаСофт, 1993. – 272 с. : ил. – ISBN 5-87458-441-2.
11. Кнут Д. Искусство программирования для ЭВМ В 2-х т. Т. 2 Получисленные алгоритмы/ Д. Кнут ; пер. с англ. – Москва : Мир, 1977.
12. Кнут Д. Искусство программирования для ЭВМ. В 2-х т. Т. 1 Основные алгоритмы / Д. Кнут ; пер. с англ. – Москва : Мир, 1976.

13. Крупник А. Ассемблер : самоучитель / Александр Крупник. – Санкт-Петербург : Питер, 2005. – ISBN: 5-469-00825-8.
14. Кубенский А. А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ / А. А. Кубенский. – Санкт-Петербург : БХВ-Петербург, 2004. – 463 с. : ил. – ISBN 5-94157-506-8.
15. Кулаков В. Программирование на аппаратном уровне : специальный справочник / В. Кулаков.–2-е изд. – Санкт-Петербург : Питер, 2003. – ISBN:
5-94723-487-4.
16. Ларман Крэг Применение UML и шаблонов проектирования / Ларман Крэг ; пер. с англ. – 2-е изд. – Москва : Вильямс, 2004. – 624 с. : ил. – ISBN
5-8459-0250-9 (рус.).
17. Лукас П. C++ под рукой / П. Лукас ;пер. с англ. – Киев : ДиаСофт, 1993. – 176 с. : ил. – ISBN 5-87554-079.
18. Олифер В.Г. Компьютерные сети. Принципы, технологии, протоколы / В. Г. Олифер, Н. А. Олифер. Санкт-Петербург : Питер, 2007. – 960 с.
19. Пауэре Л. Microsoft Visual Studio 2008 / Л. Пауэре, М. Снелл ; пер. с англ. – Санкт-Петербург : БХВ-Петербург, 2009. – 1200 с. : ил. – (В подлиннике) – ISBN 978-5-9775-0378-5.
20. Прата С. Язык программирования C++ : лекции и упражнения / Стивен Прата ; пер. с англ. – 5-е изд. – Москва : ООО И. Д. Вильямс, 2007. – 1184 с. – ISBN: 5-8459-1127-3, 0-672-32697-3.
21. Рочкинд М. Программирование для UNIX. / М. Рочкинд ; пер. с англ. – 2-е изд. перераб. и доп. –Москва : Русская Редакция; Санкт-Петербург : БХВ- Петербург, 2005. – 704 с. : ил. – ISBN 5-94157-749-4.
22. Стерлинг Л. Искусство программирования на языке Пролог / Л. Стерлинг, Э. Шапиро.–Москва : Мир, 1990.

23. Уолл Л. Программирование на Perl / Ларри Уолл, Том Кристиансен, Джон Орвант ; пер. с англ. – 3-е изд. – Санкт-Петербург : Символ-Плюс, 2002.
24. Уэзерелл Ч. Этюды для программистов / Ч. Уэзерелл ;пер. с англ. – Москва : Мир, 1982. – 288 с. : ил.
25. Фаронов В. В. Программирование на языке C#. / В. В. Фаронов. – Санкт-Петербург : БХВ-Петербург, 2007. – 240 с. : ил. – ISBN 978-5-91180-369-8.
26. Флэнаган Д. JavaScript : подробное руководство / Д. Флэнаган ; пер. с англ. – Санкт-Петербург : Символ-Плюс, 2008. – 992 с. : ил. – ISBN-10: 5-93286-103-7, ISBN-13: 978-5-93286-103-5.
27. Фридл Д. Регулярные выражения / Джеффри Фридл ; пер. с англ. – 3-е изд. – Санкт-Петербург : Символ-Плюс, 2008.
28. Хендерсон П. Функциональное программирование. Применение и реализация / П. Хендерсон ;пер.с англ. – Москва : Мир, 1983. – 349 с. : ил.
29. Шлеер С. Объектно-ориентированный анализ: моделирование мира в состояниях / С. Шлеер, С. Меллор ; пер. с англ. – Киев : Диалектика, 1993. – 240 с. : ил. – ISBN 0-13-629940-7 (англ.), ISBN 5-7707-5541-5.
30. Visual C# 2008 : базовый курс / Карл Уотсон, Кристиан Нейгел, Якоб Педерсен, Хаммер Рид Джон Д., Скиннер, Морган, Эрик Уайт ; пер. с англ. – М. : ООО "И. Д. Вильяме", 2009. – 1216 с. : ил. –(Парал. тит. англ.). – ISBN 978-5-8459-1532-0 (рус.).

Навчальне видання

Яковицький Ігор Леонідович

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

«ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ»

(для студентів освітньо-кваліфікаційного рівня бакалавр у галузі знань 12 – Інформаційні технології за спеціальністю 122 – Комп'ютерні науки та інформаційні технології)

Відповідальний за випуск І. Л. Яковицький

За авторською редакцією

Комп'ютерне верстання *І. Л. Яковицького*

План 2015, поз. 160л

Підп. до друку 07.04.2017 р. Формат 60×84/16
Друк на ризографі Ум. друк. арк. 1,2
Зам. № Тираж 5 пр.

Видавець і виготовлювач:
Харківський національний університет міського господарства
імені О. М. Бекетова,
вул. Маршала Бажанова,17, Харків, 61002
Електронна адреса: rectorat@kname.edu.ua
Свідоцтво суб'єкта видавничої справи:
ДК № 5328 від 11.04.2017 р.